UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1497

Programme 1
Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

UNIFYING RELATIONAL AND OBJECT-ORIENTED DATABASES : AN ABSTRACT DATA TYPE BASED APPROACH

Scott DANFORTH
Eric SIMON

Septembre 1991

Unification des bases de données relationnelles et orientées-objet : une approche basée sur des types abstraits de données

Scott Danforth, Eric Simon

Résumé :

Ce papier présente un modèle de données et d'opérations qui combine les approches relationnelles et orienté-objet des bases de données. Notre modèle intègre - dans un cadre simple et typé statiquement - les types de données abstraits, les objets et valeurs, les classes, les relations, le polymorphisme des méthodes et une persistance des données transparente.

Les principales caractéristiques de notre modèle sont : (i) l'utilisation d'une notion d'héritage appelée similarité de comportement qui n'impose ni l'héritage de structure, ni l'héritage de méthode comme fondation du polymorphisme de méthodes, (ii) l'autorisation d'avoir des opérations discriminées sur plusieurs arguments, et (iii) la définition de constructeurs de type classe et relation qui permettent une séparation claire entre types et collections persistantes. Nous donnons aussi des conditions suffisantes que doit satisfaire un schéma de base de données de façon à garantir un contrôle de type statique sûr des programmes.

Enfin, la puissance et la flexibilité de notre modèle sont illustrées à-travers le support de la migration d'objets dans des classes et l'héritage sur des types paramétrés en présence de mises à jour.

Mots clés : Modèle de Données, Langages de Programmation de Données, Contrôle de types, Types Abstraits de Données, Héritage, Objets Complexes, Relations.

Unifying Relational and Object-Oriented Databases : An Abstract Data Type Based Approach

Scott Danforth, Eric Simon

Abstract :

This paper presents a data and operation model that combines relational and object-oriented approaches. Our model incorporates - within a simple, statically-typed framework - support for Abstract Data Types, objects, values, classes, relations, multiple inheritance, polymorphic functions and transparent persistence.

The main features of our model are : (i) to use a notion of inheritance, called behavioral similarity, that does not require neither structural inheritance nor method inheritance as a basis for method polymorphism, (ii) to allow multi-targeted operations on ADTs (i.e., the method selection is actually done according to the run-time type of several arguments), and (iii) to define relation and class type constructors which enable a clear separation between types and persistent collections. We also describe sufficient conditions that must satisfy a database schema in order to guarantee a safe static typechecking of programs.

Finally, the power and flexibility of our data model is demonstrated via the support for object migration, and inheritance orderings on parametric types.

Keywords : Data Models, Database Programming Languages, Type Checking, Abstract Data Types, Inheritance, Complex Objects, Relations.

Unifying Relational and Object-Oriented Databases:

An Abstract Data Type Based Approach

Scott Danforth[1]
Eric Simon[2]
INRIA-Rocquencourt[3]

Abstract

This paper presents a data and operation model that combines relational and object-oriented approaches. Our model incorporates -- within a simple, statically-typed framework -- support for Abstract Data Types, objects, values, classes, relations, multiple inheritance, polymorphic functions and transparent persistence. The main features of our model are (i) to use a notion of inheritance, called behavioral similarity, that does not require neither structural inheritance nor method inheritance as a basis for method polymorphism, (ii) to allow multi-targeted operations on ADTs (i.e., the method selection is actually done according to the run-time type of several arguments), and (iii) to define relation and class type constructors which enable a clear separation between types and persistent collections. We also describe sufficient conditions that must satisfy a database schema in order to guarantee a safe static typechecking of programs. Finally, the power and flexibility of our data model is demonstrated via the support for object migration, and inheritance orderings on parametric types.

Keywords Data Models, Database Programming Languages, Type Checking, Abstract Data Types, Inheritance, Complex Objects, Relations.

1. Introduction

In this paper, we present a data and operation model that combines relational and object-oriented approaches. Our starting point is the belief that abstract data types, inheritance, shared objects, and relations are four valuable and orthogonal concepts useful in semantic modeling, and that these concepts should therefore be supported by a single database system.

Object-oriented databases [Cope84, Lecl88, Zdon90] have primarily focused on support for complex, shared objects as first class data, and the use of inheritance for defining type structures and polymorphic methods. Howewer, while these approaches address limitations of relational systems, they generally omit support for relations, views and integrity constraints, and suffer from the absence of a safe static typechecking (e.g., Ode [Agr89], Ontos [Ont90], and O2 [Lecl88]).

---

[1]MCC Systems Technology Lab, Austin, TX 78759 USA
[2]INRIA Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, FRANCE
[3]This work was performed at INRIA, supported by Esprit project STRETCH, under contract P2443.

Extensions of the relational model [Osb86, Ston88, Gard89, Kier90] have primarily focused on incorporation of abstract data types (ADTs), but have been limited by absence of direct support for sharing of data, and by absence of support for bulk data objects other than relations. Although these limitations can be surmounted procedurally, this is in conflict with the otherwise declarative nature of the relational approach. Recently, there has been a lot of research interest in supporting a declarative language by an object-oriented data model, or designing declarative, object-oriented languages [Abit89, Cac90, Kif89, Mai89, Lou91]. These data models combine objects, classes and relations, and use a type inheritance hierarchy based on structural inheritance. Howewer, as discussed in [Lou91], almost nothing is said about how methods are defined on classes and how method inheritance works. Also, the corresponding type systems (when they exist) impose strong restrictions on the use of the resulting data models.

This paper presents a new approach that is comprehensive in its support for both relational and object oriented databases, and includes in a simple statically-typed framework multiple inheritance, polymorphic methods, objects, values, classes and relations. Our approach is based on the framework provided by *Partitioned Algebras*, a formal model for object-oriented programming developped in [McKen91]. Compared to existing work, our model has three major features: (i) it uses a form of inheritance based on behavioral similarity, (ii) multi-targeted operations are allowed (i.e., the method code is selected according to the run-time type of several method's arguments), and (iii) class and relation type constructors are introduced in addition to the usual set and tuple type constructors.

The type hierarchy in our model, is not based on structural inheritance but uses instead a weaker form of inheritance called behavioral similarity. Existing object-oriented data models use different kinds of inheritance hierarchies like class extent inheritance, structural inheritance, and method inheritance. Most generally, these notions are overloaded into a single type inheritance hierarchy [Zdon90]. For instance, the following pseudo-code statements,

class *person* has attributes
    name: string;
    age: integer;

class *student* isa person has attributes
    rank: integer;

have the meaning that: a student has all attributes of person plus other specific attributes (structural inheritance), the class *person* contains all objects of class *student* (class extent inheritance), and every method defined on *person* is inherited by *student* (method inheritance). In databases, conceptual data (conceptual schema) are designed independently from their physical representation (physical schema). Merging structural inheritance and behavioral (method) inheritance into a single hierarchy makes both the conceptual and physical database schemas strongly dependent on each other. Coming back to our example, it would not be possible to represent objects of the class *student* as

class *student* has attributes
rank: integer;
have_rank: {[name: string, age: integer]}

because the structures of *person* and *student* will not be related anymore with respect to structural inheritance. On the other hand, in our model, abstract data types can be related into a behavioral inheritance hierarchy independently from their representation. Also, the class extent hierarchy is specified separately from the type hierarchy. Thus, given types:

person = [*name*: string; *age*: integer];

student = [*rank*: integer; *have_rank*: {[*name*: string; *age*: integer]}];

type *classof student* can be declared as a subtype of *classof person* in a behavioral sense, (here, we assume that *classof* is a type constructor). Furthermore, if *Sorbonne_stud* and *Paris_person* are two instances of types *classof student* and *classof person* respectively then we may additionally declare -- (but do not have to) -- that *Sorbonne_stud* is a subclass of *Paris_person*.

A second feature of our data model is the use of multi-targeted operations as opposed to single-targeted operations usually "attached" to classes. We shall see in Section 3 that this feature is essential, in particular when one wants to model views and integrity constraints as part of an object-oriented data model.

Finally, another feature of our data model is the introduction of a class type constructor that enables a clear separation between types that describe the structure of values, and classes that denote collections of values. This distinction is important because a type describes the set of all possible values of an expression (used at compile-time), whereas a class denotes a set of actual values (only used at run-time). In particular, based on such a distinction, we shall present a statically typecheckable solution to the problem of object migration in a class hierarchy, thereby providing answers to the typing issues raised by [Brea89, Alba85]. With the same objective in mind of separating types from collections of values, we also introduce a relation type constructor.

The remainder of the paper is organized as follows. Section 2 introduces useful terminology about abstract data types, operations and inheritance. Section 3 presents the main choices that guided the design of our data model, and their motivation. Section 4 describes how abstract data types can be defined, their semantics, and the inheritance hierarchy between ADTs. Operation semantics and overloading resolution are presented in Section 5. Section 6 provides examples illustrating the expressive power and flexibility of the model, including dynamic object migration and inheritance orderings on parametric types. Section 7 provides a summary and reviews our contribution.

2. Preliminary Definitions

2.1. Abstract Data Types and Types

In our model, an abstract data type is not data, but a *description* of a type of data and its semantics. An ADT specification is expressed by indicating a *representation* for instances of the ADT, and by defining *operations* that implement the desired ADT semantics for these instances.

The representation of ADT instances is indicated using a type expression; the operations are defined using programming languages. An ADT *instance* does not include the operations available on it, but is simply data that may be passed to these operations. ADTs may be declared as in the following examples:

point = [x: float, y: float]
polygon = { point }
square = [upper_left_corner: point, side_length: float]

A *point* is represented as a tuple and a *polygon* is represented as a set of *point*. The type expression used to indicate the representation of ADT instances determines what is called the *representation type* for the ADT [Mit85]. An ADT instance can either be a value or an object. Values are *anonymous* data, and objects have *identity*, i.e., no two objects are equal.

In this paper, we reserve the word *type* to indicate the denotations of type expressions. Type expressions are composed from a fixed set of type constructors and primitive types. To each type constructor corresponds a system-defined implementation structure and associated set of operations that access, add, and remove data elements according to the semantics desired for the type. The operations potentially available on an ADT instance are those that correspond to its representation type (called *representation operations*), plus those that implement the desired ADT semantics (called *semantic operations*), which are generally user-defined.

2.2. Generic Operations

We employ a traditional, procedural view of operations, thereby avoiding the message-passing metaphor in which objects are viewed as receiving messages that request a given behavior. This logical view is supported by overloaded, *generic* operations (sometimes called parametrized operations), in which the receiving object in the message-passing metaphor is simply a specially designated argument of the operation, whose runtime type is used to resolve overloading. Such an argument is called a *target* for the operation. When more than one argument type is used to resolve an overloaded operation, the operation is termed *multi-targeted*. Most current Object Oriented systems are limited to a single target, although there are exceptions (e.g., CLOS [ANSI90], and RDL/1 [Kier90]).

Operations are performed by *methods*, which are defined using a programming language (or built-in system code, for representation operations). In general, corresponding to any given operation will be a number of *operation interfaces*, each explicitly indicating the parameter ADTs for which the interface is valid. For instance, the generic operation *area* may have two interfaces: *area* (p:polygon) and *area* (s:square). Each operation interface is supported by a single method, and resolving operation overloading (either at compile-time or run-time) involves choosing the appropriate operation interface and, thereby, the appropriate method for performing the operation.

2.3. Inheritance

A partial ordering inheritance relationship between ADTs (denoted $\leq$) reflects the *opportunity* to make use of common representation and/or operational interfaces to define polymorphic method code. Given an inheritance ordering $s_1 \leq s_2$, we shall distinguish between two main interpretations

of this ordering.

A first interpretation is that the representation of the s_1 ADT is defined as a refinement of s_2's representation (we call this *representational specialization*). For instance, a tuple type can be refined by adding attributes to it or by refining the type of its attributes (see [Card84]). Thus, if *person* = [name:string] and *employee* = [name:string, sal: float] are two ADTs, then the ordering *employee* ≤ *person* is possible. Remark that this implies that every method applicable to an s_2 can work for an s_1. Assuming the ordering *employee* ≤ *person*, if *free-time* is an operation interface applicable to *person* and supported by a method *m*, then there is a corresponding interface *free-time* applicable to *employee*, supported by *m* or another method than *m*, but *m* would work anyway for *employee*.

A second interpretation is that a generic operation available on s_2 may be supported on s_1 (we call this inheritance ordering *behavioral similarity*). Behavioral similarity is based on *implementation specialization* (supporting different interfaces for a generic operation with different methods), and on *implementation similarity* (supporting different interfaces using the same *polymorphic* method). Method polymorphism is based on the use of generic operations. Given the earlier example representations for the *square* and *polygon* ADTs, an inheritance ordering *square* ≤ *polygon* is possible. The generic operation *area* can be supported on *square* and *polygon* using two different methods (implementation specialization). Furthermore, the generic operation *size_compare* that compares the area of two polygons, can be supported by a polymorphic method, i.e., the same code will apply to both a polygon and a square (implementation similarity). The reason in this case, is that squares are behaviorally similar to polygons with respect to the operation *area* used by such method. Finally, an operation interface *union* (p: polygon, q: point) can be specified to be available on *polygon* and not on *square*.

The directionality of ≤ leads naturally to *behavioral specialization*. We say that s_1 is a behavioral specialization of s_2 if $s_1 \leq s_2$, and for every operation interface applicable to an s_2, there is a corresponding interface for an s_1. Behavioral specialization is a strong condition, not particularly important in our model since it is not necessary for method polymorphism.

3. Design Choices and Motivations

We use abstract data types as the basic framework because we feel that ADTs represent a sound, well-accepted approach to data and operations applicable to both relational and object-oriented approaches [Gog87, Dan88, McKen91, Osb86, Ston88, Gard90, Lind90, Stone90, Kier90].

One can view the traditional class-based specification style used in most object-oriented languages and database systems [Mey88, Zdon90], as a useful approach to simultaneously specifying ADTs, inheritance, representational specialization (via inheritance), and behavioral specialization (via a combination of inherited methods, and class overriding default inheritance of method code). The assumption of inherited representation allows default inheritance of methods. Although classes in this sense are a concise syntactic technique for expressing ADTs and for using inheritance relations between ADTs to define ADTs incrementally, the approach is limited in generality because representational specialization and behavioral specialization become *required* as

a basis for polymorphism.

Consider the case of an engineer designing geometrical data, like squares and triangles. Their common properties (*area*, *size_compare*, *overlap*, ...) can be grouped into a more general data type, say *polygon* that both squares and triangles would inherit. This inheritance link is captured by the orderings: *square* ≤ *polygon*, and *triangle* ≤ *polygon*. Therefore, the *area* property becomes a generic operation applicable to polygons, squares and triangles. Now, pursuing his design, the engineer has to decide how these data should be represented. This is a crucial performance issue: the complexity of operations like *area* will strongly depend on these representation choices. For polygons, a reasonable data structure is a list or a set of points. But computing the area of a square like the area of an arbitrary polygon is surely not efficient. This suggests the need for a different representation for squares. For instance, if squares are represented as tuples containing a length attribute as above, then a method that implements the *area* operation will require a single multiplication. As a result, a polymorphic method, like *size_compare (p:polygon, s:polygon)*, that uses the *area* operation will be efficient when used on squares, rather than being handicapped by a representation designed for general polygons. With class-based ADT and inheritance specification, such a design will be inconsistent with the ordering *square* ≤ *polygons*, which requires that a square is a structural refinment of a polygon.

A possible solution in C++ -like languages, is to create a virtual class, say *2D-Object*, whose subclasses are *square* and *polygon*. A virtual method (i.e., a method with an empty code) *area* is then defined for *2D-Object*, and redefined for its subclasses. Finally, a method *size_compare* can be defined on *2D-Object* and inherited by its subclasses. This solution requires to define and maintain virtual classes: for instance, if *circle* is added as a *2D-Object*, a new virtual class, say *Straight_Line_2D_Object* must be created as a subclass of *2D-Object*.

Behavioral similarity represents an opportunity to tailor ADT representations in ways that are *not* based on structural refinement, but rather, the desire for operational efficiency. Therefore, we could say, anticipating our syntax for defining methods, that -- (the code for methods is omitted) --

square ≤ *polygon*;

define operation *area* (target s: square): float = ...
define operation *area* (target p: polygon): float = ...

define operation *size_compare* (p: polygon, q: polygon): polygon = ...

The argument of operation *area* is targeted (using the keyword *target*) because the method used for a given invocation of *area* depends on the type of that argument. In the last operation interface *size_compare*, there is no targeted argument because there is a single polymorphic method for any invocation of *size_compare*.

Because generic operations can be supported by compile-time as well as run-time resolution of operation overloading, according to possible target argument sorts, the only advantage offered by representational specialization is that it may allow additional opportunities for method polymorphism and compile-time operation resolution. For example, if squares *are* represented in

the same way as polygons (as a set of points), then a polymorphic method can be used to compute *area*, and, as a result, it will be possible to resolve invocations of the *area* operation within other polymorphic methods at compile-time, rather than dynamically, at run-time. This would be an important factor to take into account when choosing ADT representations, but the tradeoff between the cost of run-time method resolution and the advantage of tailoring representations for operational efficiency is not an issue to be decided a-priori by a data and operation model. Rather, we see this as an important issue for consideration by a system administrator. Based on behavioral similarity, our model is capable of supporting the advantages of representational specialization when this is desired, but does not require it as the basis for supporting polymorphic methods. Common Objects [Snyd86a, Snyd86b] provides a precedent for this approach.

Another example of the flexibility provided by behavioral similarity is the well known exception handling problem. Suppose to have a *Bird* data type with an associated operation *fly_speed*, and we wish to add a *penguin* data type. Assuming behavioral specialization, a possible solution is to create two data types *Bird_fly* $\leq$ *Bird* and *Bird_not_fly* $\leq$ *Bird*, where *Penguin* $\leq$ *Bird_not_fly* and operation *fly_speed* is defined on *Bird_fly*. This involves a reorganization of the data type hierarchy. On the contrary, behavioral similarity allows to define *Penguin* $\leq$ *Bird* and a single operation interface *fly_speed* (*target* b: Bird): integer, only available on *Bird*. There is no method for computing the fly_speed of a penguin, knowledge that will be used for typechecking methods using the *fly_speed* operation.

The second important design decision of our model is to allow multi-targeted operations. General motivations for supporting multi-targeted operations in an object-oriented language, are reported in [Snyd86b, Dan90, McKen91]. We claim that such operations are also desirable for views and integrity constraints. Below is a motivating example that relies on a usual class-based approach.

Following [Abit91], a view mechanism allows a programmer to restructure the class hierarchy of the database and modify the behavior and structure of objects. In particular, new classes can be introduced into the class hierarchy. These classes, called *virtual*, are defined by specifying their population [1]. This can be done by selecting existing objects from other classes or by creating new objects called *imaginary* objects. To populate a virtual class with imaginary objects, one specifies a query that returns a set of values. A new object identifier is then given to each value. A key point is that a virtual class should be usable as any other class.

Suppose the database has a class *Person* with subclasses *Minor* and *Adult*, and class *Beverage* with subclasses *Alcohol* and *Juices*. Consider the class *Drinks* and the virtual class *Control_drinker* populated with imaginary objects, defined as follows:

---

[1] A virtual class here has a different meaning than in C++.

class *Drinks* has attributes
drinker of type Person,
drink_name of type string,
date of type integer

virtual class *Control_drinker* has attributes
drinker of type Person,
drink of type Beverage

Now, a method *Offense* is defined on the virtual class. It checks that only adults got alcohol and computes the value of a ticket in case of offense. This method is attached to the virtual class but the method code depends on both the types of the *drinker* and *drink* attributes. A multi-targeted method like *Ticket (p:person, b:beverage)* could naturally be used within *Offense* method code. Using our notation, the operation interfaces associated with *Ticket* would be defined as follows.

define operation *Ticket* (target m: Minor, target b: Alcohol):float =
{if < (m.age, 18) 300.00
else if < (m.age, 21) 100.00}

define operation *Ticket* (target m: Minor, target b: Juice):float =
{0.00}

define operation *Ticket* (target a: Adult, target b ≤ Beverage):float =
{0.00}

If the language does not allow such methods, the methods have to be simulated with single-targeted methods, thereby making the code less readable and efficient, as shown below.

class *Minor*
method *Ticket* (b: Beverage): float =
{Tick_m (b, self)}

class *Adult*
method *Ticket* (b: Beverage): float =
{Tick_a (b, self)}

class *Juice*
method *Tick_m* (m: Minor): float = ..
method *Tick_a* (a: Adult): float = ..

class *Alcohol*
method *Tick_m* (m: Minor): float = ..
method *Tick_a* (a: Adult): float = ..

This situation arises as soon as compound objects are built by assembling existing objects, which is typically the case with virtual classes.

4. Abstract Data Types and Inheritance

We now provide a detailed description of our data and operation model. For organization, we successively present the three aspects that characterize a database in our framework: the ADTs supported by the system, an inheritance ordering for these ADTs, and a persistent database. ADTs are described by giving their names (in the case of primitive ADTs), names and representations (in the case of declared ADTs), and by giving the associated operation interfaces. The inheritance ordering is described by giving a partial ordering between pairs of declared ADTs. The persistent database is described by naming *roots of persistence*. Each such name identifies an operation of no

arguments that returns the corresponding persistent data.

In the following, we represent a *database schema* using four components: a set S of ADT-ids, a partial ordering $\leq$ representing the inheritance relationships between ADTs, a set Σ of operation interfaces, and a set R of operation names, identifying the operations in Σ that support roots of persistence. We shall denote a database schema by $\Psi = (S, \leq, \Sigma, R)$.

4.1. S -- Abstract Data Types

The ADTs include the primitive ADTs. For example, integer $\in$ S, string $\in$ S, etc. Other ADTs reflected by S are user-defined -- i.e., the result of an explicit ADT declaration expressed using a specification language that includes type expressions. The arguments to type constructors in such expressions are always ADT names because the corresponding data structures hold data that will be seen by operations, and, as explained, all data seen by operations is an ADT instance.

4.1.1. Open and Protected ADT Declarations

Two different kinds of ADTs can be declared: *open* and *protected* ADTs. An open ADT allows unrestricted access to its representation operations. For example, the following is an open tuple ADT declaration:

define type *n_tuple* = [name: string];

As a result of this declaration, the ADT-id *n_tuple* is included in S, and the operations *name_select* (n_tuple):string, and *name_assign* (n_tuple,string):n_tuple are included in Σ. Because *n_tuple* is an open ADT, any operation will be able to create n_tuples, and any operation which accepts an *n_tuple* argument will be able to use the *name_select* and *name_assign* operations on this argument.

Protected ADTs correspond more closely to the usual concept of an ADT than do open ADTs. A protected ADT declaration indicates a limited set of "privileged" semantic operations, and the representation type is then hidden from all but these operations. Because no other operation is given direct access to the representation components, the privileged operations can then collectively maintain whatever semantics is desired for the ADT. For example, the following is a protected ADT declaration.

define type *first_quadrant_point* using [x:float, y:float] visible to *mk_point*, *move*;

As a result of this declaration, the ADT-id *first_quadrant_point* is included in S, and the appropriate representation operations are added to Σ. When typechecking operations involving a *first_quadrant_point*, only the *mk_point* and *move* operations will be allowed to use its representation operations. If, for instance, it is desired that a *first_quadrant_point* should always have non-negative *x* and *y* values, the *mk_point* and *move* operations can insure this, and no other operation will be able to violate this property.

4.1.2. Object and Value ADTs

We do not want to require identity for all data in our model, and we therefore distinguish object ADTs from value ADTs. Given this decision, however, our objective is that objects and values

should be treated in exactly the same way by methods (i.e., methods should not "navigate" through data using pointers associated with objects). For example, a method for computing the area of a square should work on all squares -- whether they are objects (and thus potentially shared), or values. As shown below, this result is reflected in the behavioral inheritance hierarchy relating ADTs.

The set S of ADT-ids can be divided into four partitions: $\mathbf{S} = \mathbf{P} \cup \mathbf{D} \cup \mathbf{O} \cup \mathbf{V}$, where **P** names primitive ADTs (e.g., *integer*, *string*, *float*, etc.), **D** names declared ADTs (e.g., *polygon*, *square*, *person*, *employee*, etc.), where $\mathbf{O} = \{\text{obj } s \mid s \in \mathbf{P} \cup \mathbf{D}\}$ names object ADTs (e.g., *obj integer*, *obj polygon*, etc.), and $\mathbf{V} = \{\text{val } s \mid s \in \mathbf{P} \cup \mathbf{D}\}$ names value ADTs (e.g., *val integer*, *val polygon*, etc.).

Thus, to any primitive or declared ADT corresponds an implicitly defined object ADT and an implicitly defined value ADT. The *obj* and *val* prefixes used in object and value ADT-ids are not type constructors, but simply the initial portions of implicitly defined ADT names. All runtime data is either an object or a value, and explicitly identified as such by its ADT name. Objects are created by applying the operation *new* to data. For example, evaluating the expression *new (new (<point>[x:1.0, y:3.14]))* creates two, distinct new objects -- each is an instance of the *obj point* ADT, and each has a data attribute storing a tuple with the indicated *x* and *y* values. There can be no sharing between these objects because their data attributes contain only values (in particular, *val float* instances). Thus, updating one of these objects (perhaps by "moving" it to a new *x,y* location) cannot affect the other object.

As mentioned above, object and value ADTs are related by the inheritance ordering. If *s* is a primitive or declared ADT, then *obj s* and *val s* are also ADTs. In addition, $\textit{obj } s \leq s$, $\textit{val } s \leq s$, and, if $s \leq t$, then $\textit{obj } s \leq \textit{obj } t$ and $\textit{val } s \leq \textit{val } t$. These behavioral relationships are guaranteed by the system.

4.1.3. Type Constructors

In addition to the usual tuple and set type constructors, relation and class type constructors are included in the model because sets do not completely support the semantics we desire for relations and classes. A notion of class data type which is just a bulk data type, like sets and lists, has also been independently proposed in [Ghe90], in order to avoid any confusion between a class and a type. Our support for classes and relations as data structures is motivated by the following considerations. Classes with subclasses provide a useful indexing technique for objects, and relations are useful to represent application-level associations between data. Because classes and relations are first class data structures in our model, they can be included within ADT representations, and can be persistent. The O2 data model [Lecl89] takes a somewhat similar approach with respect to classes; in that system, classes do not define representations or operations, but are named, persistent data structures used to hold objects.

We assume the types denoted by tuple and set type constructors are familiar, and do not review their use here. Section 4.1.4 formalizes tuples and sets with respect to their interpretations in the model.

Classes Classes are provided as types in order to support subclasses, and to provide representation operations for modification of classes by adding or removing subclasses, and for computing class extents. Also, the *insert* operation semantics for classes differs from that of sets: inserting a value into a class creates an object.

Classes hold only objects. If *s* names a primitive or declared ADT, the expression *classof s* denotes a class type, capable of holding the representations of ADT sorts s' ≤ obj s. For example, given the ADTs *polygon*, and *square*, the following class ADTs may be declared:

define type *polygon-class* = classof polygon;

define type *square-class* = classof square

As a result of this declaration, the open ADT names *polygon-class* and *square-class* are included in S, and the appropriate representation operations on these ADTs are included in Σ. If *square* ≤ *polygon*, then squares as well as polygons can be added to a *polygon-class* instance *p*, and a *square-class* instance *s* can be added to the (originally empty) set of subclasses for *p*. The *extent* operation returns a set (not a class) including all objects in a class, plus, recursively, all objects in the extents of its subclasses. Thus, if *s* is a subclass of *p*, the extent of *p* will include all objects in *s* as well as those of *p*.

Relations Relations are provided as types in order to support a traditional model of relations as a homogeneous set of tuples, each holding attributes of specified ADT domains. In contrast, a set of type { *t* } can hold elements of any sort t' ≤ t in the behavioral inheritance hierarchy.

Just as classes hold only objects, relations are restricted to holding value ADTs -- in particular, open value ADTs represented using tuples. Given an open declared ADT of sort *t*, where *t* is represented using a tuple, the expression *relationof t* denotes a relation type, capable of holding representations of the ADT sort *val t*. For example, assuming that *employee* and *project* ADTs are declared, the following ADT declarations are possible:

define type *emp-tuple* = [e: obj employee, p: obj project];

define type *emp-relation* = relationof emp-tuple;

As a result of this declaration, the ADTs *emp-tuple* and *emp-relation* are included in S, and the appropriate operations on *emp-tuple* and *emp-relation* are included in Σ.

4.1.4. ADT Interpretations

In addition to the structure indicated by the type expression, the representation type for an ADT in our model includes structure for holding information used by the system: an ADT-id, and an object-id. If the type expression indicated in an ADT declaration is τ, then the representation type for the ADT is a tuple of type [sort:ADT-id, id:object-id, data:τ]. All ADT instances in our model are represented using such a tuple, and all operation arguments are typed as ADT instances. There are no user-level operations for accessing the *sort* and *id* information held by ADT instances -- this information is visible to and used only by the system and by system-defined methods.

An ADT interpretation is a set containing all possible instances of an ADT. Figure 1 gives interpretations for all possible ADTs. We express ADT interpretations using tuples, sets, and elements of atomic semantic value domains -- those corresponding to the primitive types (i.e., the sets of atomic elements commonly named integer, boolean, string, etc.), plus the two following sets of atomic elements: object-ids O = {o1, o2, ...}; and ADT-ids S = {s1, s2, ...}. We use ϕ to represent either a primitive or declared ADT name, assuming ADT declarations of the form $\phi = \tau$ where τ is a type expression built from ADT names and any of the type constructors: tuple, set, relationof, and classof. We assume O includes a distinguished element named *null*. We denote the interpretation of an ADT of sort s using $[\![s]\!]$, and the interpretation of a type expression τ using $[\![\tau]\!]$. The interpretation of a primitive type τ (e.g., integer) is the set of atomic elements named by τ.

$[\![\phi]\!] = \{ [\text{sort}:\phi, \text{id}:o, \text{data}:d] \mid o \in O, d \in [\![\tau]\!] \}.$

$[\![\text{val } \phi]\!] = \{ [\text{sort}:\text{val } \phi, \text{id}:\text{null}, \text{data}:d] \mid d \in [\![\tau]\!] \}.$

$[\![\text{obj } \phi]\!] = \{ [\text{sort}:\text{obj } \phi, \text{id}:o, \text{data}:d] \mid o \in O\text{-}\{\text{null}\}, d \in [\![\tau]\!] \}.$

$[\![[A_1:\phi_1, ..., A_n:\phi_n]]\!] = \{ [A_1:v_1, ..., A_n:v_n] \mid v_i \in [\![\phi'_i]\!], \phi'_i \le \phi_i, 1 \le i \le n \}, \quad n > 0.$

$[\![\{\phi\}]\!] = \{ S \mid v \in S \Rightarrow$ there exists $\phi' \le \phi$ such that $v \in [\![\phi']\!] \}.$

$[\![\text{relationof } \phi]\!] = \{ R \mid v \in R \Rightarrow v \in [\![\text{val } \phi]\!] \}, \quad \phi$ an open tuple

$[\![\text{classof } \phi]\!] = \{ [\text{objects}:O', \text{subclasses}:S] \mid$
$o \in O' \Rightarrow$ there exists $\phi' \le \text{obj } \phi$ such that $o \in [\![\phi']\!]$, and
$s \in S \Rightarrow$ there exists $\phi' \le \phi$ such that $s \in [\![\text{classof } \phi']\!] \}.$

Figure 1 -- ADT Interpretations

4.2. $\le$ -- The Inheritance Ordering As mentioned during the discussion of value and object ADTs, some inheritance orderings are implicit. Other explicitly declared orderings among the declared ADTs are possible, such as the following:

employee $\le$ *person*	*regular_polygon* $\le$ *polygon*
student $\le$ *person*	*rectangle* $\le$ *polygon*
student_employee $\le$ *student*	*square* $\le$ *regular_polygon*
student_employee $\le$ *employee*	*square* $\le$ *rectangle*

Although inheritance declarations for ADTs are arbitrary, they have important typechecking implications for the methods used to support operations on these ADTs. For example, if it is desired to define a polymorphic method valid on any person, and this method uses the operation

free-time, then, given the above inheritance declarations, a *free-time* operation must be available for the ADTs *person*, *student*, *employee*, and *student-employee*.

5. Operations, and Persistence

5.1. Σ -- The Operation Interfaces

To each operation interface in Σ will correspond a single method for supporting the desired semantics of the operation and returning the appropriate result. When appropriate, a single *polymorphic* method can be used to support multiple interfaces of an operation. As mentioned earlier, many operations and their methods are built-in -- i.e., provided automatically by the system in support of the primitive ADTs and access to the representation structure of declared ADTs. Other interfaces in Σ result from explicit method definitions, as discussed in Section 5.1.4, below.

5.1.1. Representation and Semantics of Operation Interfaces

We represent Σ as an indexed collection of sets of operation interfaces. The index into Σ is the operation name, and each interface specifies argument sorts, targets, and a result sort. The following example illustrates this. We assume that *small_point_set* ≤ *point_set*, *large_display_screen* ≤ *display_screen*, and *square* ≤ *polygon*. For purpose of illustration, we include in this example three different interfaces for a *mk_polygon* operation.

---

mk_polygon: (point_list, display_screen): polygon
(*target* point_set, large_display_screen): polygon
(*target* small_point_set, display_screen): polygon

Figure 2 -- Example Operation Interfaces

The semantics of the operation interfaces is based on whether or not operation arguments are targets (denoted by the *target* keyword). If an operation interface is not targeted on a particular argument, then it (and the method implementing the operation) must be polymorphic in this argument. If an operation interface is targeted on a particular argument, then any polymorphism of the operation with respect to this argument must be explicitly indicated using distinct interfaces. The reasoning behind this approach is that because non-target arguments are not used to resolve operator overloading, operators must be polymorphic with respect to such arguments for any given set of target ADTs. On the other hand, since (by definition) target arguments are used to resolve overloading, operation polymorphism is not required with respect to target arguments, and should not be assumed. Section 5.1.3, below, describes the process of method selection that is based on this semantics.

The first *mk_polygon* operation interface in Figure 2 is not targeted, which means that the method supporting this interface must be polymorphic with respect to both the kinds of *point_list*

and the kinds of *display_screen* used as its arguments (i.e., any first argument sort $p \leq$ *point_list* must be acceptable as the first argument, and similarly with respect to *display_screen* for the second argument).

5.1.2. Coherent Database Schema

We require that it be possible to statically guarantee that programs execute without runtime type errors (i.e., it must be possible to statically guarantee that all overloaded operation applications in a program will be successfully resolved). The formal model presented in [McKen91] provides constraints (called *properness*, *regularity* and *consistency*) that must be placed on operation signatures in order to guarantee meaningful behavior and support static typing. Following [McKen91], we require that a database schema satisfies these constraints and call it a *coherent* database schema.

A database schema is proper if intersecting interfaces of generic operations have the same target arguments. Let $m\ \omega_1$:s and $m\ \omega_2$:s' be two operation interfaces [1] if there exists an ω_0 such that $\omega_0 \leq \omega_1$ and $\omega_0 \leq \omega_2$ (by comparing the individual ADT names of the ω's pairwise), then the interfaces $m\ \omega_1$:s and $m\ \omega_2$:s' are said to *intersect*. This requirement is weaker than requiring all interfaces of a given operation to have the same targets, but is still sufficient to support static typing. Assuming there is no inheritance relationship between *point_list* and *point_set*, there is no conflict between the untargeted *mk_polygon* operation and the targeted versions of the operation (i.e., the schema is proper).

Regularity means that given a particular operation and set of argument ADTs, there will either be *no* entry in Σ describing the operation for these ADTs (i.e., there is no available method for performing the desired operation), or there will be a *single* most applicable method for performing the operation on these ADTs. Regularity prevents ambiguities with respect to operation overloading. Because the *mk_polygon* operation with interface *mk_polygon* (target point_set, large_display_screen) is targeted on its first argument, an explicit entry for the *mk_polygon* operation with interface *mk_polygon* (target *small_point_set*, *display_screen*) is required to indicate the existence of a method for supporting this interface. Absence of an explicit entry would indicate that there is no method for building a polygon from a *small_point_set* -- knowledge that would be important for typechecking methods using the *mk_polygon* operation.

Consistency means that if an interface x of an operation is applicable to more specialized target arguments than another interface y of the operation, then the result type of x is more specialized than that of y, and the non-target arguments of y are more specialized than those of x. Consistency represents a generalization of the traditional ordering on function types to a multi-targeted context [Dan88], and provides an essential basis for correct typechecking of polymorphic methods [McKen91, Dan90]. In the third *mk_polygon* interface, as required by consistency, the non-target *display_screen* argument was not specialized (in fact, it was generalized).

---

[1] $m\ \omega_1$:s is of the form m (s1, ..., sn):s.

5.1.3. Resolving Operation Overloading

Given an application of an operation on a set of arguments, an interface for the operation is found in Σ such that the sorts of the target arguments specified in the operation interface exactly match those of the corresponding invocation arguments. An operation invocation that has been typechecked with respect to a coherent database schema is guaranteed to have exactly one such interface in Σ (due to regularity). For example, an application of the m*k-polygon* operation typechecked with respect to the operation interfaces of Figure 2 would be handled as follows:

The method supporting the untargeted *mk_polygon* operation is chosen if:
(1) the first argument ADT $p \leq$ *point_list*, and
(2) the second argument ADT $s \leq$ *display_screen*.

The method supporting the first of the targeted *mk_polygon* operations is chosen if:
(1) the first argument ADT $p \leq$ *point_set*, and
(2) not ($p \leq$ *small_point_set*), and
(3) the second argument ADT $s \leq$ *large_display_screen*.

Otherwise, the method supporting the final *mk_polygon* interface is chosen.

Given that the operation application has been typechecked, there are no other possibilities. One of the above three methods will be chosen to implement the operation.

5.1.4. Method Definitions

The non-system operations described by Σ appear as a result of ADT operation *method definitions*. These provide the method code whose purpose is to implement the operation interfaces. In this section, we make no assumptions concerning the language used to express methods, other than the necessity for indicating the types of parameters for which a method is intended.

<op-def> ::= define operation <op-id> [(<arg-list>)]:<result> = <method-code>
<arg-list> ::= [target] <arg-spec> [{, [target] <arg-spec>}...]
<arg-spec> ::= <param-id> : <ADT-id>
| <param-id> ≤ <ADT-id>
| <ADT-id> < <param-id> ≤ <ADT-id>
<result> ::= <ADT-name>
| like <param-id>

e.g., define operation *mk_polygon* (p ≤ point_list, s ≤ display_screen):polygon = ...
define operation *mk_polygon* (p : target point_set, s ≤ large_display_screen):polygon = ...
define operation *mk_polygon* (p : target small_point_set, s ≤ display_screen):polygon = ...

Figure 3 -- Method Definition Syntax

To show how this information might be provided, Figure 3 presents and illustrates an informal grammar for operation definitions. Within this grammar, nonterminal symbols are enclosed in angle brackets <>; alternative productions are introduced with |; [*a*] means that *a* is optional; and {*a*}... means that *a* is repeated one or more times.

Following the grammar in Figure 3 are example method definitions for supporting the three *mk_polygon* interfaces introduced in Figure 2 (the method code is omitted). As can be seen from the argument specifications in these examples, each method is polymorphic with respect to various arguments. Method code, expressed using a programming language, must be typechecked to assure that it supports the interfaces corresponding to its definition, and that all operations used by the method will be successfully resolved.

5.2. R -- The Persistent Roots

A persistent root is an object with identity, and the data returned by the operation corresponding to such a root is an instance of this object. The method supporting this operation is system-defined code provided for establishing addressability of persistent data within programs. Specific commands must be available for creating persistent data. In the following, we only consider persistent classes and relations.

---

define type *square_class* = classof square;

create class *Districts* of type classof polygon;
create class *Windows* of type square_class;
create class *Gardens* of type square_class subclass of Districts;

define type *rel_part* of type relationof [part: string, sub: string];

create relation *Parts* of type rel_part;
create derived relation *Subpart* of type rel_part;

Figure 4 -- Persistent Class and Relation Definition

A persistent class is created by applying the command *create class* to a class ADT. Similarly, the command *create relation* can be applied to a relation ADT for creating a persistent relation. Examples are given on Figure 4.

Classes and relations are always created empty. A set of parents may be specified when a class instance is created; the resulting class/subclass relationship is required to obey the inheritance ordering on the class element ADTs (as reflected in the ADT interpretations of Figure 1). The first *create class* command of Figure 4, creates an object named *Districts* whose value is an empty class of type *classof polygon*. Corresponding to this object, there is an operation *Districts* () with no arguments that returns the current value of the persistent root (the actual class extent). Because,

Gardens is declared to be a subclass of class *Districts*, the *Districts* () operation will return all the objects in *Districts* and *Gardens*.

Two additional keywords, *base* and *derived*, can be used in class and relation creation statements. When a *base* persistent root is created, it is extensionally defined (i.e., its elements are the result of explicit insert operations). By default, the keyword *base* is always assumed. On the other hand, a *derived* persistent root is intensionally defined (i.e., its elements are dynamically computed by invoking the corresponding extent operation). In the example of Figure 4, relation *Subpart* is intensional, and represents the transitive closure of the *Part* relation. This requires to define an operation, say *mk_Subpart ()*, which will be automatically invoked by the *Subpart ()* operation when the value of *Subpart* is required. Assuming a language with an imperative style for defining operations, Figure 5 gives a possible implementation of the *mk_Subpart ()* operation. We assume the existence of an operation *Subpart_Part_compose ()*, which returns the composition (join followed by project) between relations *Part* and *Subpart*.

```
define operation mk_Subpart () =
local lastr = new (<rel_part> {});
begin
  % copies Parts into Subpart %
  foreach tuple in Parts ()
        Subpart + <rel_part> [part:tuple.part, sub:tuple.sub];
  endfor
  % computes the closure of Subpart %
  while <> (lastr, Subpart) % tests if new tuples have been generated %
        begin
        lastr := Subpart;
        foreach tuple in Subpart_Part_compose ()
          Subpart + <rel_part> [part:tuple.part, sub:tuple.sub];
        endfor
        end
  endwhile
end;
```

Figure 5 -- Definition of an Intensional Relation

Formally, a *database instance* is modeled using two functions: π, and ν, where π is a *root assignment*, and ν is a *object store*. A root assignment π corresponds to the operations in Σ that return persistent roots. For any $r \in \mathbf{R}$, $\pi(r)$ is the data associated with the persistent root named r. An object store ν maps each element of an ADT interpretation domain into an element of the same domain. Like in IQL [Abit89], the purpose of ν is to reflect object-sharing by mapping all objects with the same identity to the same semantic value. If x is any data item, we call $\nu(x)$ the *current representation* of x. This is the only one that is seen by operations. For values, ν is the identity

function. For objects, the indirection with respect to object-ids provided by ν is essential to supporting general graph structures within data.

An auxiliary function ρ (defined in terms of ν) is also introduced in order to map classes to class extents. Figure 6 summarizes this definition.

A root assignment π maps each name in **R** to data as follows:
If $r \in \mathbf{R}$, and r is reflected in Σ as being of sort ϕ, then $\pi(r) \in [\![\phi]\!]$.
An object store ν maps data to data as follows:
If $x \in [\![\text{val } \phi]\!]$, then $\nu(x) = x$
If $x \in [\![\text{obj } \phi]\!]$, then $\nu(x) \in [\![\text{obj } \phi]\!]$, and $x.id = \nu(x).id$
If $x \in [\![\text{obj } \phi]\!]$, $y \in [\![\text{obj } \phi]\!]$, and $x.id = y.id$, then $\nu(x) = \nu(y)$.
The extent function ρ is defined on an arbitrary ADT instance d as follows:
If $d \in [\![\text{classof } \phi]\!]$ for some ϕ,
then $\rho(d) = \nu(d).data.objects \cup \{ o \mid o \in \rho(C), C \in \nu(d).data.subclasses \}$
otherwise, $\rho(d) = d$.

Figure 6 -- Database Instance

---

6. Features of the Data and Operation Model

To provide examples suggestive of the expressive power and flexibility of the model, we now discuss support for object migration, and inheritance orderings on parametric types.

6.1. Object Migration

Suppose to have a database with types and classes for persons, students and employees (we assume here a standard class-based approach):

class *Persons* has attributes
name: string;

class *Students* isa *Persons* has attributes
id: integer;
has: {course}

class *Employees* isa *Persons* has attributes
id: string;
sal: integer;

Suppose that an object of *Persons*, as a result of applying to a University, becomes a student, while retaining *person* behavior. In a traditional class-based ADT specification approach, because classes and types are not distinguished, this requires to change the type (i.e., the class) of the object of *Persons*. A trivial solution would be to delete the object from class *Persons* and add a new object into *Students*. The new object will get a different identity than the original object, because an object of *Persons* cannot be seen as an object of *Students* (i.e., a student cannot be assigned with a person). This change of identity is a problem because it requires to change every

reference to the original object in the database into a reference to the new one. What we need is that the *same* oid-value which represented a person, after the operation, with the same identity, represents a student. This updating operation is usually called *object migration.*

Different solutions to this problem have been proposed in the usual class-based framework. In [Cac90, Kim90], an object with the original oid-value is inserted in the destination class and the original object is unchanged. Thus, the information about an object (identified by its oid-value) may be disseminated in several classes. For instance, if a person *o*, becomes a student, then a new object with same identity than *o* is inserted into *Students*. The typing constraint enforced by the system is to verify that migration is performed downward the inheritance hierarchy (i.e., from classes to subclasses). Howewer, there are two main problems with this approach.

First, accessing the whole information about an object requires to access all classes containing this object. This "composition" operation represents a significative overhead, even if it is optimized using indices based on oid-values. Second, the migration operation could yield run-time errors. Suppose that the person which is intended to become a student has already been specialized into an employee, with a structure incompatible with student (e.g., the *id* attribute is of a different type). In this case, the value produced for the person who becomes a student and is also an employee, is not correctly typed for all data structures which still refer to it as an employee (e.g., the *id* field is expected to be of type *string*). In order to avoid this problem, we could assume the existence of a class *Student_employees*, subclass of both *Students* and *Employees*, and then migrate the employee to this class so that he also acquires the behavior of *Students*. But, suppose that a *free_time* operation is defined on *Employee* and *Student_employees* with two different methods. If *free_time* is invoked with an object of the class *Employee*, then one must know if the object is also in *Student_employees* in order to select the appropriate method code. This seriously complicates the overloading resolution procedure.

A solution to the above problems is proposed in [Ghe90]. First, like in our data model, classes are separated from types by introducing a specific class type constructor. Therefore, the migration of a person to a student essentially requires to change the type of the object in the class *Persons* into a *student* type and then to move the object from one class to the other. To fix the ideas, we introduce the following types:

person = [name:string];
student = [name:string, id: integer, has:{course}];
employee = [name:string, id:string, salary:float];

student ≤ *person*; *employee* ≤ *person*;

Classes *Persons*, *Students* and *Employees* are of respective type *person*, *student*, and *employee*. A special operation, called *specialize*, is defined to transform an object of type *s* with an oid-value *o* into an object of type *t* with the same oid-value *o*, provided that $t \leq s$. In order to decide if a type specialization operation can be accepted, a knowledge of all the other specialization operations which have been applied to the object is needed. Let us define the *proper types* of an object as consisting of all the types resulting from each specialization operation applied to it, plus the type with which it has been created. In our previous example, the proper types of a person becoming an

employee are *person* and *employee*. A type rule then checks that the new proper type of an object must be a subtype of all the old proper types. This means that every object has a minimum proper type and that the new minimum proper type must be a subtype of the old minimum proper type. This rule is required in order to avoid *structural* incompatibilities between types just as described above. For instance, if a person is now intended to be a student, then the new proper type is clearly not a subtype of both *person* and *employee*. Howewer, as discussed in [Ghe90], there is no way of determining statically the minimum proper type of an object. This means that run-time typechecking is involved thereby entailing a cost overhead.

Our data model allows object migration through the ADT hierarchy via *down* and *up* operations, whose behavior is described below. These operations provide a facility much like a cast operation in conventional languages (e.g., C++); they may (or may not) actually change the current representation of an object, but in any case, under the control of safe static typing, the object they return can be treated differently than the object they are given as an argument.

The *down* and *up* operations are given two arguments: an object (say, *objdata* of "current" sort *obj o*), and data (say, *newrep* of "destination" sort either *val d* or *obj d*). They are targeted on the first argument; *Down* (resp. *up*) then performs as follows. If the destination sort *d* is strictly below (resp. above) *o* in $\leq$, then a new current representation for *objdata* is created in which the sort attribute is *obj d* and the data attribute is that of *newrep*; otherwise the current representation of *objdata* is unchanged. *Objdata* is then returned as the result of the operation. During typechecking, the result sort for the operation is the object sort corresponding to *newrep*.

It is impossible to use *down* in a way that violates the type system because any data structure in our model that can contain an object of sort s can also contain an object of sort $s' \leq s$, and typechecking is already concerned with all such s'.

Coming back to our previous example, suppose that an *obj person* instance *o* is intended to become an *obj employee* instance. This is performed by invoking: down (*o*, *obj employee*). Assuming that the representation type of *o* was:

[sort: *obj person*, id: o1, data: [name: Marie]]

then it would be transformed into:

[sort: *obj employee*, id: o1, data: [name: Marie, id: eng101, sal: 50,000]]

Suppose that *o* is intended to become a student. Then, because of the semantics of *down*, we require the existence of a type, say *student_employee* = [name:string, id:string, course:string], defined to be a subtype of both *employee* and *student*. The *down* operation can be invoked with *down* (o, obj student_employee). The new representation type of *o* will be:

[sort: *obj student_employee*, id: o1, data: [name:Marie, id:worker1005, course:CS]

Thus, if object *o* above is intended to move to the *Employees* class, its representation type is first changed using the *down* operation and then the object (actually, its representation type) will be deleted from *Persons* and inserted in *Employees*. During this migration operation, the only type rule that needs to be enforced for the *down* operation is that the destination sort is below the sort of the object given as argument. Unlike [Ghe90], this rule can always be enforced statically and

efficiently. The simplicity of our type rule is only due to the flexibility given by the use of an inheritance hierarchy based on behavioral similarity.

The *up* operation is useful when learning more about an object suggests the need to remove behavior inherited from some higher sort (e.g., a *student_employee* may quit his job while remaining a *student*). A first step in dealing with this situation is to remove the object from any data structures capable of containing only objects with the undesired inherited behavior (e.g., if the *student_employee* quits his job, he should be removed from classes whose objects must be of sort $\leq$ *obj employee*). This can be done independently of changing the object's sort.

Having done this, however, there are two reasons for using *up* to replace the current representation of an object with one appropriate to a higher destination sort. First, in the case where an operation is available in both the destination sort and the current sort, the methods supporting the operation for these sorts may be different (e.g., a *free_time* method for *student* may be different than that for *student_employee*). If the current representation of the object is not changed using *up*, even though the method appropriate for the destination sort (i.e., *student*) is desired, the method appropriate on the current sort (*student_employee*) is the one that will be used. Secondly, it may be desired to use *down* to specialize the object in some new way, but even if there is a path from the object's destination sort to this new sort, there may be no such path from the object's current sort. (For example, assuming tutors cannot be employees, the student may have quit his job to become a tutor.) For these two reasons, *up* is provided.

If the first step identified above has been undertaken, then using *up* to migrate the object upwards cannot violate the type system and *up* will succeed. In order to assure type safety, however, *up* must verify that this first step has indeed been taken -- this involves examining all data structure components in the system capable of holding sorts below the destination sort, and verifying that none of these structures contain the object. Only if this test is satisfied can *up* replace the current representation of the object with one appropriate to the destination sort. If, on the other hand, the test is not satisfied, then *up* must signal a runtime error, because there is no way to proceed without the possibility of violating of the type system.

We emphasize that a failure for *up* is not a type error -- it prevents type errors -- and is entirely reasonable given the circumstances under which it can happen. The correct response to such an error is to remove the object from the data structures within which it does not belong, after which *up* can be used successfully.

6.2. Inheritance for Parametric Types

Consider two ADTs represented using tuples containing a *shape* attribute, where it is desired that both the ADTs and their *shape* attributes be related by $\leq$. Figure 7 illustrates two such ADTs and the corresponding *shape_assign* operation interfaces that are provided by the system. Certainly, these interfaces must be targeted on their first arguments; in general, houses and buildings may be represented quite differently (in this example they have a single attribute in common), and it would be overly restrictive to assume that a single method for performing this operation is suitable in both cases.

As for targeting on the second argument, there are important reasons for this as well. First, it is required by consistency since *house* ≤ *building*, and *rectangle* ≤ *polygon*. Also, for example, depending on the implementation of tuples, the actual method code used to implement *shape_assign* interfaces for (*building*,*polygon*) and (*building*,*square*) could be different. Different structures might be copied (because squares might be represented differently that polygons), so the code for performing this copying might differ. On the other hand, perhaps only a pointer would be copied, in which case the code for implementing both interfaces might be the same. But these possibilities are not reflected at the level of operation interfaces, which assume nothing about implementation.

S *building* = [..., shape:polygon, ...]
house = [..., shape:rectangle, ...]

≤ *house* ≤ *building*
square ≤ *rectangle* ≤ *polygon*

Σ *shape_assign*: (target building, target polygon):building
(target building, target rectangle):building
(target building, target square):building
(target house, target rectangle):house
(target house, target square):house

Figure 4 -- Representation Operations

Aside from the targeting of the *shape_assign* interfaces that are provided by the system, it is important to note that there is no *shape_assign* interface provided for (*house*,*polygon*). This is not only reasonable, given the representation of a *house*, but imperative for safe typechecking, which is ultimately based on representation types. If such an operation interface were provided, it would allow a violation of the type system. This example demonstrates a solution to an open problem of inheritance orderings for parametrized types (which is what type constructors are) in the presence of updates.

The usual expression of this problem goes as follows. If *square* ≤ *polygon*, then is Array(*square*) ≤ Array(*polygon*)? In the absence of updates, it is safe to use this ordering, but if updates are allowed, then this natural and useful ordering cannot be used [Born82]. This paradox arises in the presence of updates because a procedure that types a parameter as Array(*polygon*) might, if allowed to receive an Array(*square*) as an argument, update this array argument by storing a *polygon* in it, thus violating the semantics of Array(*square*). Graver's response to this problem [Grav89] is to use different orderings for parametric types, depending on whether updates are part of the programming language used to express methods. Our feeling is that this approach really misses the central point: that a particular (update) operation should simply be absent because

it is not part of the semantics of the parametric types that are involved. It is not possible to view things this way from the perspective of a class-based specification style, because inheritance is equated with behavioral specialization.

The flexibility necessary to support inheritance orderings for parametric types is available in our model as a result of supporting multi-targeted operations, and using $\leq$ to reflect behavioral similarity rather than behavioral specialization. As a result, the desired ordering can be available for use in our model (we can define a polymorphic method that works for general buildings, including houses), and, at the same time, a straightforward approach to typechecking will prevent a misuse of this ordering in the case of updates (a method that might potentially require a *(house,polygon) shape_assign* interface will be refused by the typechecker because there is no such interface). This solves a serious open problem for object oriented systems by extending the utility of safe static typechecking to a context including updates and useful, intuitive inheritance orderings on parametrized types.

7. Conclusion

We have presented a data and operation model that unifies the concepts of object-oriented databases and relational databases extended with abstract data types into a simple and statically-typed framework. Our data model is based on the framework provided by *Partitioned Algebras*, a formal algebraic model for object-oriented programming developped in [McKen91]. Compared to existing work, our data and operation model has three major features. First, inheritance ordering is based on behavioral similarity. This enables to tailor ADT representations in ways that are not based on structural refinment, but rather, the desire for operational efficiency. It also provides flexibility to deal with exception handling by not imposing behavioral specialization as a consequence of declared inheritance. Second, multi-targeted operations are allowed. This is very useful to model methods on views represented by virtual classes as described in [Abit91]. Finally, class and relation type constructors are introduced in addition to the usual set and tuple type constructors. This enables a clear separation between types and persistent collections.

The impact of our contribution is twofold: first, we extend object-oriented databases by providing typesafe and satisfactory solutions to the problems of object migration and orderings on parametric types, while avoiding explicit pointer-based navigation. Second, we extend relational approaches by supporting sharing in the presence of updates to objects, and by supporting relations within a context including general-purpose programming languages and ADT domains ordered by inheritance.

Acknowledgements

The ideas related to multi-targeted operations are the result of conversations with Bob McKenzie. We wish to express our appreciation for these discussions, and for permission to refer to his work. We want to thank Stefano Ceri and Letizia Tanca for their helpful and constructive criticism of earlier versions of the model. Special thanks are due to Dennis Shasha for the fruitful discussions we had, and careful reading of a preliminary version of this paper.

REFERENCES

[ANSI90] ANSII X3J13 Standards Committee, "Common Lisp Object System Reference," 1990.

[Abit89] S. Abiteboul, and P. Kanellakis, "Object Identity as a Query Language Primitive," Proceedings ACM SIGMOD Int. Conf., Portland, 1989.

[Abit91] S. Abiteboul, and A. Bonner, "Objects and Views", Proceedings ACM SIGMOD Int. Conf., Denver, May 1991.

[Alba85] A. Albano, L. Cardelli, and R. Orsini, "A Strongly Typed Interactive Conceptual Language", ACM Transactions on Database Systems, Vol.10, N.2, 1985.

[Agr89] R. Agrawal, and N. H. Gehani, "ODE (Object Database and Environment): The Language and the Data Model", Proc. ACM-SIGMOD Int. Conf., Portland, Oregon, May 1989, pp. 36-45.

[Born82] A. Borning, and D. Ingalls, "A Type Declaration and Inference System for Smalltalk," Proceedings of the 9th ACM Symposium on Principles of Programming Languages, 1982.

[Brea89] V. Breazu-Tannen, P. Buneman, and A. Ohori, "Can Object-Oriented Databases be Statically Typed?", 2nd International Workshop on Database Programming Languages, Portland, Oregon, June 1989.

[Cac90] F. Cacace, S. Ceri, S. Crespi-Reghizzi, L.Tanca, R. Zicari, "The Logres Project: Integrating Object-Oriented Data Modeling with a Rule-Based Programming Paradigm," Proceedings of ACM SIGMOD Int. Conf., Atlantic City, May 1990.

[Card84] L. Cardelli, "A Semantics of Multiple Inheritance", in *Semantics of Data Types*, G. Kahn, D. McQueen, G. Plotkin Eds., LNCS 173, Springer Verlag, 1984.

[Cope84] G. Copeland, and D. Maier, "Making Smalltalk a Database System," Proc. of ACM SIGMOD Int. Conference, Boston, May 1984.

[Dan88] S. Danforth, and C. Tomlinson, "Type Theories and Object-Oriented Programming," ACM Computing Surveys, Vol. 20, No. 1, March, 1988.

[Dan90] S. Danforth, "Multi-targeted Virtual Functions for OODB," Proceedings of the 6th Conf. on Advances in Databases, Edited by INRIA, Montpellier, France, Sept. 1990.

[Gard89] G. Gardarin, et al., "Managing Complex Objects in an Extensible Relational DBMS," Proceedings of the 15th Int. Conf. on VLDB, Amsterdam, 1989.

[Gard90] G. Gardarin, and P. Valduriez, "ESQL: An Extended SQL with Object and Deductive Capabilities", Proc. Int. Conf. on Databases and Expert Systems Applications, Vienna, Austria, Aug. 1990.

[Ghe90] G. Ghelli, "A Class Abstraction for a Hierarchical Type System," Proceedings of the Third Int. Conference on Database Theory, Springer Verlag LNCS No. 470, 1990.

[Gog87] J. Goguen, and J. Meseguer, "Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics," In *Research Directions in Object-Oriented Programming*, B. Shriver and P.Wegner, Eds., MIT Press, 1987.

[Grav89] J. Graver, "Type-Checking and Type-Inference for Object-Oriented Programming Languages," Ph.D. Dissertation, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1989.

[Kier90] G. Kiernan, C. de Maindreville, and E. Simon, "Making Deductive Databases a Practical Technology: a Step Forward," Proceedings ACM SIGMOD Int. Conf., Atlantic city, May 1990.

[Kif89] M. Kifer, and J. Wu, "A Logic for Object Oriented Programming (Maier's O-Logic Revisited)", Proc. of Int. Conf. on PODS, March 1989.

[Kim90] W. Kim, N. Ballou, J. Garza, and D. Woelk, "Architecture of the Orion Next-Generation Database System", IEEE Trans. on Knowledge and Data Engineering, Vol 2, N. 1, March 1990.

[Lecl88] C. Lecluse, P. Richard, and F. Velez, "O2, An Object-Oriented Data Model," Proceedings ACM SIGMOD Int. Conf., Chicago, May 1988.

[Lecl89] C. Lecluse, et al., "The O2 Database Programming Language," Proceedings of the 15th Int. Conf. on VLDB, Amsterdam, 1989.

[Lind90] B. Lindsay, and Laura Hass, "Extensibility in the Starburst Experimental Database System," IBM Almaden Research Report RJ 7570 (70747) 7/5/90, 1990.

[Lou91] Y. Lou, and M. Ozsoyoglu, "LLO: An Object-Oriented Deductive Language with Methods and Method Inheritance", Proceedings of ACM SIGMOD Int. Conf., Denver, May 1991.

[McKen91] B. McKenzie, "An Algebraic Model of Object-Oriented Programming," Ph.D. dissertation, University of Texas at Austin, Computer Science Dept., To appear.

[Mey88] B. Meyer, "Object-Oriented Software Construction", Computer Science, Prentice Hall, 1988.

[Mit85] J. Mitchell, and G. Plotkin, "Abstract Types have Existential Type," Proceedings 12th ACM Symposium on Principles of Programming Languages, 1985.

[Ont90] Ontologic Corporation, "Ontos, Product Description", March 1990.

[Osb86] S. Osborn, and T. Heaven, "The Design of a Relational Database System with Abstract Data Types for Domains," ACM Transactions on Database Systems, Vol. 11, No. 3, September, 1986.

[Snyd86a] A. Snyder, "Encapsulation and Inheritance in Object-Oriented Languages," Proceedings of the ACM Conf. on Object-Oriented Systems, Languages and Applications, Portland, Oregon, 1986.

[Snyd86b] A. Snyder, "CommonObjects: An Overview", ACM SIGPLAN Notices, Vol. 21, N. 10, Oct. 1986, pp. 19-29.

[Ston88] M. Stonebraker et al., "Inclusion of New Types in Relational Database Systems," Proc. Int. Conf. on Data Engineering, 1988.

[Ston90] M. Stonebraker, et al., "Third-Generation Data Base System Manifesto," Memorandum No. UCB/ERL M90/28, College of Engineering, University of California, Berkeley, 1990.

[Zdon90] S. Zdonik, and D. Maier (Eds.), "Fundamentals of Object-Oriented Databases," in *Readings in Object-Oriented Database Systems*, Morgan-Kaufman, San Mateo, CA., 1990.

ISSN 0249-6399